PERGAMON

Applied Mathematics Letters 15 (2002) 969–973

Applied Mathematics Letters

www.elsevier.com/locate/aml

A Constraint Logic Programming Approach for Generating All Perfect Matchings

F. HARARY
Department of Computer Science
New Mexico State University
Las Cruces, NM 88011, U.S.A

G. GUPTA
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75083, U.S.A.

(Received September 2001; revised October 2001)

Abstract—In [1,2], a heuristics based approach is presented for finding all perfect matchings of a graph. We present a simpler and more elegant approach based on constraint logic programming that embodies the same heuristic.

1. INTRODUCTION

As in [3], a *graph* $G = (V, E)$ consists of a finite nonempty set V of *nodes* and a subset E of the set of all two-subsets of V, i.e., pairs of distinct nodes, called *edges*. Let $\mathtt{n} = |V|$ and $\mathtt{m} = |E|$. Two *adjacent nodes* are joined by an edge. Two *adjacent edges* have just one common node. Two edges are *disjoint* if they are not adjacent. A *matching* of G is a set of pairwise disjoint edges. A *perfect matching* (PM) contains all the nodes. Obviously, if G has a PM then $\mathtt{n}$ is even. The problem of finding all perfect matchings of a graph G has received considerable attention [1,2,4]. The problem of finding a single perfect matching can be solved in polynomial time. However, finding all the perfect matchings requires exponential time. Our objective is to present a constraint logic programming solution to finding all perfect matchings.

In [1,2], Balakrishnan and Venulingam present an approach based on artificial intelligence in which a heuristic based search is used to find all the perfect matchings of a graphs. Their heuristic picks the node with the *greatest lower degree* (GLD) [1], i.e., the node with "minimum number of first neighbors and maximum number of second neighbors". An edge incident with this node is next selected, and placed in the disjoint set of edges found so far. If the selected edge has a common node with some of the edges in the current matching, then another edge is selected. This step is repeated until a perfect matching is found. Our purpose is to present a simpler and more elegant approach based on *constraint logic programming* [5,6] that embodies the same heuristic. We refer to the heuristic based on selecting a node with greatest lower degree the *GLD heuristic*.

0893-9659/02/$ - see front matter Typeset by $\mathcal{A}\mathcal{M}\mathcal{S}$-TEX
PII: S0893-9659(02)00071-X

2. CONSTRAINT LOGIC PROGRAMMING

Logic programming [7] is a declarative programming paradigm based on a subset of first-order logic called Horn-clause logic. The best known logic programming language is Prolog [8]. The execution model used by logic programming traditionally relies on:

(i) *SLD-resolution*—intuitively, used to solve each atomic formula using the clauses in the program; and,

(ii) *unification*—intuitively, used to solve equations between logical terms to determine the applicability of a clause [7].

We assume that the reader is familiar with logic programming. Variables in traditional logic programming are assumed to range over the (possibly infinite) set of all terms that can be constructed given the constants and function symbols that occur in the logic program. Constraint logic programming (CLP) with finite domains (FD) [5,6] is an extension of logic programming in which certain variables are restricted to range over a finite set of values (called the domain of that variable), and this finite set is indicated in the program. We will refer to CLP with finite domains by the generic name CLP(FD), e.g., CHIP [5] is a CLP(FD) system. The CLP(FD) technology has been shown to be very useful for a variety of practical applications. It uses the information regarding the domain of variables and constraint propagation techniques such as *forward checking* and *look ahead* to considerably prune the search space. A typical ordinary logic program has the general form `generate & test`, that is, goals that generate a potential solution precede the goals that test that generated solution. In contrast, a constraint logic program sets up (and solves to the extent possible) the test goals (constraints) first, which is followed by the generation phase. Early constraint solving is made possible due to knowledge about domains of variables. For example, given that a variable `X` has the set `{1, 3, 5}` as the domain over which its value can range, then the constraint $\texttt{X} \neq \texttt{3}$ can be solved by narrowing the domain of `X` to `{1, 5}`, even though the actual value of `X` is not known. As a result of this early setting up and solving of constraints, the search space in a constraint logic program is considerably reduced, resulting in much faster program execution [5]. We assume that the reader is familiar with CLP(FD). Expositions of CLP(FD) can be found in [5,6].

3. A CLP SOLUTION

Our goal is to find a set of `n/2` disjoint edges. Let us label the edges in the graph $G(V, E)$ by the numbers 1 to `m`. Let us denote the labels of those edges that constitute a perfect matching by variables `E1,...,Ek`. Each of the variables `E1,...,Ek` has as its domain the set `{1,...,m}`. Our task is to choose a value for each of `E1` through `Ek` from their respective domains such that any two edges `Ei` and `Ej`, $\texttt{1} \leq \texttt{i,j} \leq \texttt{m}$, are disjoint.

In our approach, a graph is represented by coding its line graph [3] as a logic program. Thus, if the edge labeled 1 in the line graph is adjacent to edge labeled 2 , then this will be represented by the CLP fact `adjacent(1, 2)`. Given that the symmetrical fact `adjacent(2, 1)` represents the same information, without loss of generality, we will use the representation in which the first argument of adjacent is numerically smaller than the second argument. Thus, `adjacent(1,2)` represents the adjacency of edge labeled 1 with edge labeled 2.

The set of edges that constitutes a perfect matching will be ordered by the number which is used to represent them. This is done to avoid reporting symmetrical solutions. (However, this implicit ordering can also be encoded in the adjacency information; see later.)

Thus, the structure of the CLP(FD) program, using CHIP [5] notation, to compute all perfect matchings is as follows.

```
find_matching([E1,...,Ek]) :-
                E1..Ek in 1..m,
                E1 < E2 < ... < Ek,
```

```
            disjoint([E1,...,Ek]),
            deleteffc([E1,...,Ek]).

disjoint([X|L]) :- notadj(X,L), disjoint(L).
disjoint([]).

notadj(X, [Y|L]) :- not(adjacent(X,Y)), notadj(X, L).
notadj(X,[]).
```

The `disjoint` predicate holds if any two of the edges `E1,...,Ek` are adjacent to each other. The first statement in the body of `find_matching` sets up the domains of all the variables `E1` through `Ek` to `1..m`, where `m` is the number of edges in the graph $G(V, E)$. The next two statements state the constraints that should hold for `E1` through `Ek` to constitute a perfect matching. The final predicate in the body of `find_matching`, `deleteffc` is the generator of values for the variables `E1` through `Ek`.

The generator `deleteffc` is needed because once the domains and constraints have been set up, the next task is to assign values to the variables from their respective domains after they have been narrowed to the maximum extent possible as a result of constraint propagation. CLP(FD) systems provide a number of standard built-in predicates for choosing the order in which variables will be instantiated and the order in which the suspended constraints will be reinvoked. We choose the `deleteffc` builtin in which the variable with the smallest domain is chosen first, and in case of a tie the variable that is constrained the most (i.e., it appears in most suspended constraints) will be instantiated first [5,6].

The heuristics that one gets by using the `deleteffc` builtin of CLP(FD), is similar to the GLD heuristics of [1,2]. However, our approach based on CLP is more elegant, since the solution is stated declaratively. The CLP(FD) program is much shorter, compared to the one presented in [1,2], yet it achieves the same effect. The GLD heuristics of [1,2] picks the node with "minimum number of first neighbors and maximum number of second neighbors", [1,2] and then places an edge emanating from this node in the current perfect matching. The edges that are adjacent to this selected edge are deleted from the graph, the next node with minimum first neighbors and maximum second neighbors is found in the resulting graph and the process continues until the perfect matching is found. If at any point, the procedure fails then it systematically backtracks and finds another candidate edge. This continues until all the edges of the perfect matching are found. In the constraint logic program above, the heuristic used in the `deleteffc` builtin ensures that the edge that has the least number of adjacent edges is chosen first. This heuristic will lead to a solution faster, because an edge that has a larger number of adjacent edges is more likely to cause a failure since adding this edge to the perfect matching will cause all its adjacent edges to be taken out of consideration.

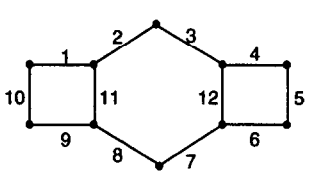

Figure 1. An example graph.

We reproduce the complete program for the example in Figure 1. (The example graph is taken from [1].)

```
perfect_matching([E1,E2,E3,E4,E5]) :-
        E1 in 1..12,
        E2 in 1..12,
        E3 in 1..12,
```

```
        E4 in 1..12,
        E5 in 1..12,
        notalladj([E1,E2,E3,E4,E5]),
        deleteffc([E1,E2,E3,E4,E5]).

notalladj([X|L]) :- notadj(X,L), notalladj(L).
notalladj([]).

notadj(X, [Y|L]) :- isnotadj(X,Y), notadj(X, L).
notadj(_,[]).

isnotadj(1,X) :- X in (3..9)\/{12}.
isnotadj(2,X) :- X in (4..10)\/{12}.
isnotadj(3,X) :- X in 5..11.
isnotadj(4,X) :- X in 6..11.
isnotadj(5,X) :- X in 7..12.
isnotadj(6,X) :- X in 8..11.
isnotadj(7,X) :- X in 9..11.
isnotadj(8,X) :- X in {10}\/{12}.
isnotadj(9,X) :- X = 12.
isnotadj(10,X) :- X in 11..12.
isnotadj(11,X) :- X = 12.
```

Note that the program is a little different from the template shown previously, in that instead of coding adjacency information for edges, we indicate which edges are not adjacent to which other edges via the `isnotadj` predicate. In the `isnotadj` predicates, for each edge numbered h, we indicate those edges that are not adjacent to h and that are numbered higher than h. Thus,

$$\texttt{isnotadj(1,X)} : -\texttt{Xin(3..9)\/12}$$

states that the edge labeled 1 is not adjacent to edges labeled 3 through 9 or 12. This results in a clearer program and also allows us to avoid including the check `E1 < E2 < < E3 < E4 < E5`, since this becomes implicit in our encoding.

The above program has been run on the Sicstus Prolog system [9] to compute the perfect matching of the graph of Figure 1 to produce all four solutions. Note that this program is specific to the graph in Figure 1. It is quite straightforward to write a metaprogram that will accept a given graph as input and automatically generate the specific constraint logic program for that graph.

4. CONCLUSION

We presented a constraint logic programming (CLP) solution to the problem of finding all perfect matchings. Constraint logic programming is a paradigm of programming based on a subset of first-order logic and constraints. It permits elegant specification and efficient solution of many combinatorial and graph theoretic problems. The constraint logic program for finding all perfect matchings is quite succinct, naturally uses the first-fail heuristic of choosing an edge in the perfect matching that is constrained the most, and we believe, is simpler and more elegant than the approach presented in [1,2].

REFERENCES

1. M.M. Balakrishnarajan and P. Venuvanalingam, Heuristic enhancements of the search for the generation of all perfect matchings, *Appl. Math. Lett.* **9** (2), 49–53, (1996).
2. M.M. Balakrishnarajan and P. Venuvanalingam, An artificial intelligence approach for the generation and enumeration of perfect matchings on graphs, *Computers Math. Applic.* **29** (1), 115–121, (1995).

3. F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, (1969).
4. K. Fakuda and T. Matsui, Finding all the perfect matchings in bipartite graphs, *Appl. Math. Lett.* **7** (1), 15–18, (1994).
5. P. Van Hentenryck, *Constraint Handling in Prolog*, MIT Press, Cambridge, MA, (1988).
6. K. Marriot and P. Stuckey, *Programming with Constraints*, MIT Press, Cambridge, MA, (1997).
7. J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, Heidelberg, (1986).
8. L. Sterling and E.Y. Shapiro, *The Art of Prolog*, MIT Press, Cambridge, MA, (1994).
9. M. Carlsson, G. Ottosson and B. Carlson, An open-ended finite domain constraint solver, *Proc. Programming Languages: Implementations, Logics, and Programs*, (1997).